Hybrid Code-Data Prefetch-Aware Multiprocessor Task Graph Scheduling

Morteza Damavandpeyma[1], Sander Stuijk[1], Twan Basten[1,2], Marc Geilen[1] and Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Embedded Systems Institute, Eindhoven, The Netherlands
{m.damavandpeyma, s.stuijk, a.a.basten, m.c.w.geilen, h.corporaal}@tue.nl

***Abstract*—The ever increasing performance gap between processors and memories is one of the biggest performance bottlenecks for computer systems. In this paper, we propose a task scheduling technique that schedules an application, modeled with a task graph, on a multiprocessor system-on-chip (MPSoC) that contains a limited on-chip memory. The proposed scheduling technique explores the trade-off between executing tasks in a code-driven (i.e. executing parallel tasks) or data-driven (i.e. executing pipelined tasks) manner to minimize the run-time of the application. Our static scheduler identifies those task sequences in which it is useful to use a code-driven execution and those task sequences that benefit from a data-driven execution. We extend the proposed technique to consider prefetching when choosing a suitable task order. The technique is implemented using an integer linear programming framework. To evaluate the effectiveness of the technique, we use an application from the multimedia domain and a synthetic task graph that is used in related work. Our experimental results show that our scheduler is able to reduce the run-time of an MP3 decoder application by 8% compared to a commonly used heuristic scheduler.**

***Keywords*-Code generation, run-time minimization, scratch-pad memory, scheduling, ILP**

I. Introduction

Multiprocessor Systems-on-Chip (MPSoCs) are used to fulfill the increasing demand for computational performance of emerging applications. MPSoCs offer a promising solution to the ever-increasing digital electronics market desire for more sophisticated and integrated applications. Nowadays, MPSoCs are used in different electronic devices such as consumer appliances, medical and navigation systems, industrial equipment, etc. Most of these devices run applications that perform complex processing operations on streams of input data. The performance of these devices depends on the performance of the processing elements as well as on the performance of the memory system. Processor performance has been improving by 60% per year [1]. However, memory access times have improved by less than 10% per year [1]. The resulting performance gap between processor and memories encouraged designers to put more effort into this crucial issue. On-chip memories have been introduced to alleviate this issue. These memories limit the number of off-chip (remote) memory accesses. On-chip (local) memory can be used as caches or as scratchpad memories (SPMs). SPMs have become an efficient replacement for caches in novel embedded systems, thanks to their lower energy/area cost and better predictability [2]. Due to limitations in the size of local memories (i.e. SPMs), an application (code and data) can only be partially loaded to the SPM. Therefore, applications must be split into several smaller tasks. Instead of loading a whole application to local memory, which requires a large memory space, tasks are loaded consecutively one-by-one based on their scheduling order. After the completion of one task, its code can be discarded from the local memory and its data can be written back to the remote memory such that free space is created for another task.

In this paper, we propose a scheduling technique for a task graph which is mapped to an MPSoC. The proposed technique tries to minimize the time needed to execute the task graph. To achieve this goal, the technique determines a task execution order that minimizes the total access time on the local and remote memories. It does this by considering the ratio between the code size of the tasks and the amount of data that is needed by the tasks. Data-intensive tasks [3] will be scheduled in a data-driven manner and code-intensive tasks [3] will be scheduled in a code-driven manner.

We implement the proposed scheduling technique using an integer linear programming (ILP) framework. The technique generates a compact ILP formulation. We apply our technique on an application from the multimedia domain (an MP3 decoder) and on a synthetic task graph from a closely related paper. The experimental results show that our technique reduces the run-time of the MP3 decoder and the synthetic task graph by 7% and 29% respectively as compared to a commonly used technique.

Nowadays, MPSoCs are equipped with direct memory access (DMA) controllers. DMA was devised to liberate processors from transferring data between different memories in a memory hierarchy. Using a DMA, the transfer of code or data to/from a local on-chip memory and the execution of a task on a processor can be overlapped. As a second contribution, we show how our scheduling technique can be extended to take prefetching into account during the task scheduling. In contrast to scheduling techniques that consider prefetching opportunities in a post-processing step after construction of the task schedule, our technique is able to find schedules that make better use of prefetching. The experimental results show that our extended technique reduces the run-time of the MP3 decoder and the synthetic task graph by 8% and 32% respectively as compared to a commonly used technique that considers prefetching in a post-processing step.

The remainder of this paper is organized as follows.

This work is supported in part by the Dutch Technology Foundation STW, project NEST 10346.

Sec. II discusses related work on task scheduling in MPSoCs. Sec. III presents a motivating example. Sec. IV describes our target MPSoC platform. Sec. V introduces our application model. Sec. VI describes the proposed scheduling technique. Sec. VII extends the proposed technique with the notion of prefetching. Sec. VIII contains an experimental evaluation. Conclusions are drawn in Sec. IX.

II. Related work

There is a rich literature on mapping applications onto MPSoCs [4]–[6]. Mostly, this work proposes mapping algorithms followed by a scheduling technique to meet design constraints such as performance, energy consumption, communication cost, or memory usage. In this paper, we rely on existing mapping methods like the one proposed in [6] and we focus on the scheduling problem. We explore two groups of related work. The first group considers the trade-off between code and data in a pipelined parallel system during task scheduling. The second group considers code and data prefetching while scheduling the tasks.

As mentioned before, our scheduling technique considers the trade-off between the code-driven and data-driven execution of parallel applications that need to be scheduled on an MPSoC. A similar problem is studied in [7] where the authors propose an evolutionary algorithm to find an optimal schedule for pipelined parallel task graphs. The algorithm does not consider the effect of memory operations. Memory access times have a large impact on the performance of streaming applications. For this reason, and in contrast to [7], we consider memory access times in our scheduling technique. In [8], a technique is presented to generate a data-parallel schedule from applications modeled as synchronous dataflow graphs. Similar to [7], the authors do not consider the overhead of moving code and data between on-chip and off-chip memory. Furthermore, they implicitly assume the availability of unlimited on-chip memory. Our scheduling technique alleviates both of these issues.

In [9], a prefetching and partitioning technique is presented to minimize the execution time of nested loops by iteratively re-timing the instructions of the loops. Their solution maps the instructions of the nested loops to multiple processing units with the objective of increasing parallelism. It only considers the effect of partitioning on the prefetching efficiency. It does not consider the effect of scheduling on the prefetching efficiency in a single partition. As compared to [9], one aim of our work is to find a suitable schedule in order to reduce the run-time of applications by maximizing the amount of the possible code/data prefetching in MPSoCs.

A large amount of research exists on optimizing SPM accesses. In [10], a heuristic is presented to partition variables of an application such that they can be mapped to the on-chip memories of an MPSoC; the heuristic performs task scheduling while considering the effect of scheduling on the variable partitioning. They assume that the time needed to access the off-chip memory can be hidden completely through prefetching. In our work, we consider the situation in which off-chip accesses are not negligible. The authors of [11] investigate the use of prefetching for SPM memories. Based on profiling information, they add software prefetching commands inside the application source code to prefetch instructions. Our work is different from this type of work because we use prefetching at a task level granularity to prefetch both code (instructions) and data. Furthermore, our technique result in a predictable solution because it is computed based on the task graph dependency relation and not based on a technique like profiling that may cause miss-predictions during the prefetching phase.

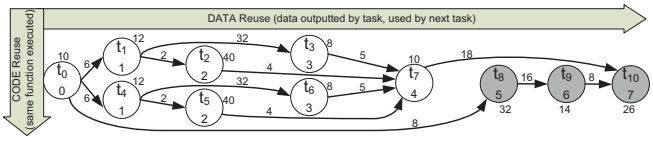


Figure 1. Example task graph.

The most relevant work to our paper is [12] which proposes an ILP-based solution to map an application modeled with a task graph onto a Cell processor. The authors assume that the application code fits in the local memory and they solve the problem of mapping data objects to memories. Task scheduling is left to run-time and no design-time analysis is suggested in [12]. Our technique provides a design-time approach for the task scheduling problem; it finds an efficient task order to minimize the run-time of the applications without incurring any run-time overhead. The authors of [12] compare their technique with a few well-known heuristics. We evaluate our techniques using the same heuristics.

III. Motivation and Preliminaries

We assume that each processor has its own local memory to store code and data. A task can start its execution on a processor if its code and input data are available in the local memory and when there is enough space in the local memory to store all output produced by the task. We also assume that a DMA unit is available for each processor.

We show the effect of different scheduling strategies and prefetch-aware scheduling on the performance of an application when running on an MPSoC with a simple example. Fig. 1 shows a task graph of an artificial application. The number close to a task is the code size of the task and the number close to an edge is the size of the data that needs to be communicated between the tasks. The number inside each task represents the function of the task. The tasks with the same function number have the same code. For simplicity, we assume in this example that all tasks have the same execution time. Assume that tasks t_0-t_7 are mapped to one processor and the remaining tasks are mapped to another processor. For the sake of brevity, we only discuss the scheduling of tasks t_0-t_7. Consider the situation in which the processor, which has to execute these tasks, has 40KB of local code memory and 40KB of local data memory. In this work, we assume that the remote memory access times have a linear relation with the amount of memory objects to be transferred (i.e. if transferring 100 bytes takes x time-units, then transferring 1000 bytes takes $10x$ time-units).

Fig. 2 shows four alternative schedules for our example task graph. Schedule A is a code-driven schedule with

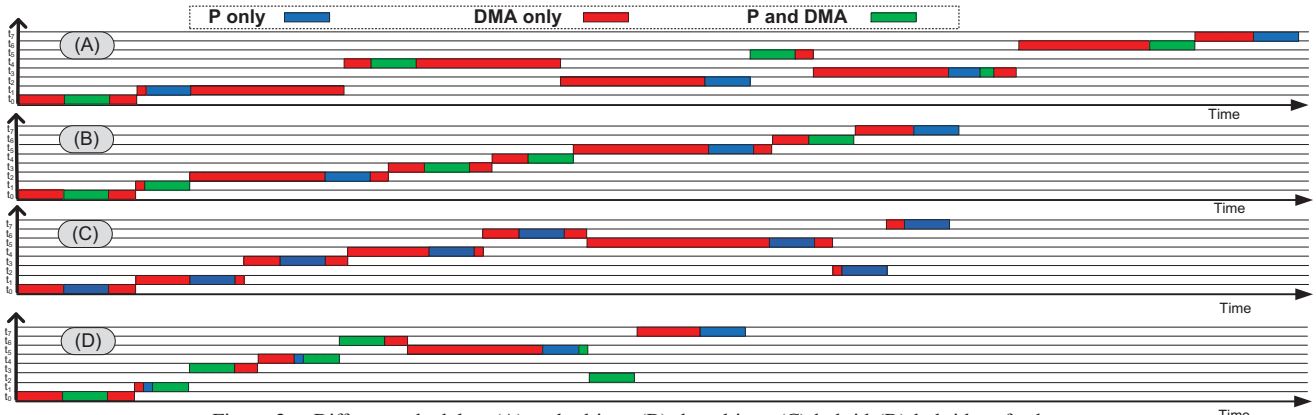


Figure 2. Different schedules: (A) code-driven (B) data-driven (C) hybrid (D) hybrid prefetch-aware.

prefetching, B is a data-driven schedule with prefetching, C is a combined code and data driven (hybrid) schedule without prefetching, and D is a hybrid prefetch-aware schedule. The blue bars in Fig. 2 indicate when a processor (P) is busy executing a task. The red bars show the activation of DMA. The green bars indicate that the processor and DMA are active simultaneously. Schedules C and D are constructed using the scheduling techniques proposed in this paper. The goal of this paper is to find schedules that minimize the run-time of applications. To realize this objective, our scheduling strategy explores alternative schedules that use a combination of code and data-driven scheduling while taking the impact of prefetching into account.

In Fig. 1, the horizontal arrow shows the direction of consecutive pipelined tasks and the vertical arrow show the direction of parallel tasks that may use the same code (i.e. execute the same function). The tasks in our example task graph can be executed in a *code-driven* or *data-driven* manner. In a code-driven schedule, the code needed for subsequent tasks will be reused as much as possible. Hence, the code needs to be loaded only once from remote memory. In a data-driven schedule, the data needed for subsequent tasks can remain in the local on-chip memory. In other words, scheduling tasks in a data-driven manner avoids moving data between the local and remote memory. Schedule C in Fig. 2 uses a combination of both code and data-driven scheduling. Tasks t_2 and t_5 are scheduled in a code-driven manner while tasks t_0, t_1, t_3, t_4, t_6, and t_7 are scheduled in a data-driven manner. In general, a code-driven strategy gives better performance for tasks that have a large code size and a data driven strategy gives better performance for tasks that operate on large data objects.

Schedule D is an extension of schedule C that is optimized for prefetching. It can be seen in Fig. 2 that schedule D is able to keep the processor and DMA active simultaneously for a longer period of time compared to schedules A and B (i.e. the total size of the green bars in schedule D is larger than the total size of the green bars in schedules A and B). This example shows that the order in which tasks are scheduled may have a noticeable impact on the amount of code and data that can be prefetched. This, in turn, has an impact on the overall completion time of the schedule.

IV. MPSoC Platform Template

Fig. 3 shows the (abstract) MPSoC platform template that is targeted in this work. The platform template consists of a set of processing tiles (PTs) that are interconnected through a shared bus or network-on-chip. Each processing tile contains a processor (P), a code memory (CM), a data memory (DM), and a direct memory access (DMA) unit. The DMA unit can work independently from the processor and it has direct memory access on the CM and DM as well as the remote memory. A real world example of this type of architecture is the Cell processor. Each processing tile is specified by a pair $pt_i = (mc, md)$ where mc specifies the capacity of the code memory (in bytes), md specifies the capacity of the data memory (in bytes). We use a constant H to model the per-word read/write latency of the remote memory in terms of clock cycles. Without loss of generality, we assume that the local memory can be accessed within one clock cycle. This is similar to the assumption made in [13]. We also assume that the DMA units of all tiles can work in parallel with each other without causing interference on the interconnect and remote memory. Partitioning remote memory into multiple banks is a common solutions [14] to realize this assumption.

V. Application Graph

An application is modeled with an acyclic task graph $G = (T, E)$ where T is the set of tasks and E is the set of dependencies between these tasks. Let $|T|$ denote the number of tasks in the task graph. We assume that the mapping of tasks to processing tiles is given. Each task is specified with a 4-tuple $t_i = (m, c, \tau, f)$ where m specifies the processing tile to which the task is mapped, c is the code size of the task (in bytes), τ is the execution time of the task (in cycles), and f is the function identifier of the task. Tasks that have the same function identifier require the same code to be executed. When such tasks are executed immediately after each other on the same processor, then we only need to load the code of these tasks prior to the execution of the first task. Each dependency edge between two tasks is specified with a 4-tuple $e_{ij} = (t_i, t_j, d, s)$ where t_i is the source task, t_j is the sink task, d is the communication delay in cycles and is equal to the number of required processor clock cycles to transfer a data object from one local memory to another local memory, s is the amount of memory (in

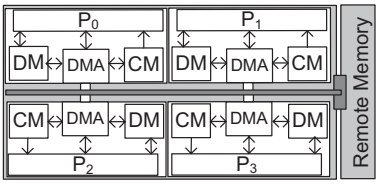


Figure 3. Target MPSoC platform template.

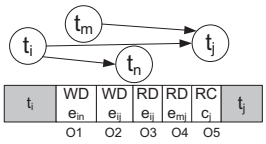


Figure 4. Initialization steps.

bytes) required in the DM to store the data of this edge. This data corresponds to the intermediate data between the source task and sink task that are connected to the edge e_{ij}.

VI. Hybrid Task Scheduling

In this section, we provide an integer linear programming (ILP) formulation to solve the scheduling problem. The proposed scheduling technique explores the trade-off between executing tasks from the task graph in a code or data-driven manner. The scheduler identifies tasks with a large code size and consecutive tasks with a large communication data size. The scheduler choose a proper scheduling order for all tasks in the task graph to minimize the overall execution time of the application. We call the proposed technique Hybrid ILP (HybILP). All necessary elements to form the ILP formulation are explained in the following subsections.

A. ILP Variables

We consider two groups of ILP variables in our formulation. The first set (S) models the start times of the tasks in the task graph. These start times also form the final solution of the scheduling problem. The second set (I) captures the ordering of the tasks that are running on the same processor.

S_i start time of task t_i

$$I_{ij} = \begin{cases} 1 & \text{if task } t_j \text{ scheduled immediately after task } t_i \\ 0 & \text{otherwise} \end{cases}$$

Based on the set I (described above), we define a notation to model the initiation time (IT) of a task which is the time needed to complete all required operations before the task can be executed. Fig. 4 shows a simple task graph with four tasks. Assume that task t_j will be executed immediately after task t_i on the same processor. The following operations are necessary before the execution of task t_j:

WD_{ij} Write the necessary output data of the tasks executed before task t_j to the remote memory (O1 and O2 in Fig. 4)

RD_{ij} Read the necessary input data of task t_j from the remote memory (O3 and O4 in Fig. 4)

RC_{ij} Read the necessary code of task t_j from the remote memory (O5 in Fig. 4)

In some situations, it is not necessary to perform all these initialization operations. Analysis of the dependencies between tasks reveals when certain operations can be skipped:

- It is unnecessary to read/write the intermediate data between consecutive tasks from/to the remote memory, when the intermediate data can be used immediately by the next scheduled task (i.e. O_2 and O_3 in Fig. 4).
- It is unnecessary to load the code of a task from the remote memory if its function is the same as the previously executed task since the code of the task already exists in the local memory (O_5 in Fig. 4 can be skipped if the t_i and t_j execute the same function).

Assume that task t_i and t_j execute the same function. The initialization time of task t_j, i.e. IT_j, is then equal to:

$$IT_j = \texttt{(Time of } O_1\texttt{) + (Time of } O_4\texttt{)} \tag{1}$$

In general, the initiation time of a task t_j is equal to:

$$IT_j = \sum_{i=0}^{|T|-1} I_{ij} \cdot IT_{ij} \tag{2}$$

IT_{ij} is the initialization time of task t_j when its previous task is known to be task t_i. It can be computed as follows:

$$IT_{ij} = H \cdot (WD_{ij} + RC_{ij} + RD_{ij}) \tag{3}$$

IT_{ij} is computed by multiplying the size of the memory objects to be transferred to/from the remote memory (i.e. summation of WD_{ij}, RD_{ij}, and RC_{ij}) with the latency of the remote memory (H). In the remainder of this subsection, it is shown how each part of Eqn. 3 can be computed. We assume that the size of the tasks in the task graph and the size of the local memories are of the same order of magnitude. In other words, the code size of the tasks and the intermediate data between the tasks are assumed to be similar in size. Therefore, we do not consider the situation in which the memory objects of two different tasks can be placed in the local memory at the same time. These assumptions are inline with the objective of this paper, which is optimizing the memory behavior of an application running on an MPSoC with limited local memories.

1) Size of the output data: The size of the output data produced by the task t_i which is needs to be written to the remote memory is given by the next equation. It is assumed that task t_j is scheduled immediately after task t_i.

$$WD_{ij} = \underbrace{\sum_{e_{i,k} \in E} \texttt{[s of } e_{i,k}\texttt{]}}_{\alpha} - \underbrace{\texttt{[s of } e_{i,j}\texttt{]}}_{\beta} \tag{4}$$

In Eqn. 4, α is equal to the size of all output data produced by the task t_i and β is the size of all data produced by task t_i and used by task t_j. The ILP solver could decide to schedule tasks that communicate large data objects in a consecutive order to get a lower initialization time (i.e. a data-driven scheduling strategy could be selected).

2) Size of the code: The size of the code needed to be fetched from memory to execute task t_j, which is scheduled immediately after task t_i, is as follows:

$$RC_{ij} = \begin{cases} 0 & \text{[f of } t_i\text{] = [f of } t_j\text{]} \\ \texttt{[c of } t_j\texttt{]} & \text{otherwise} \end{cases} \tag{5}$$

Eqn. 5 could force the ILP solver to schedule large tasks with the same functionality in a consecutive order (i.e. a code-driven scheduling strategy could be selected).

3) Size of the input data: The size of the input data that needs to be read from remote memory to execute task t_j, which is scheduled immediately after task t_i, is as follows:

$$RD_{ij} = \underbrace{\sum_{e_{k,j} \in E} [\texttt{s of } e_{k,j}]}_{\alpha} - \underbrace{[\texttt{s of } e_{i,j}]}_{\beta} \quad (6)$$

In Eqn. 6, α is equal to the size of all input data needed for task t_j and β is equal to the intermediate data between task t_i and task t_j. As before, the ILP solver could decide to schedule the tasks t_i and t_j immediately after each other if their intermediate data is large (i.e. a data-driven scheduling strategy could be selected).

B. ILP Constraints

This subsection introduces the constraints that are used in our ILP formulation. These constraints force the ILP solver to satisfy essential properties of the application (i.e. data dependencies) and intrinsic properties of the scheduling problem (i.e. avoid resource conflicts).

1) Data dependency constraints: To model the dependencies between tasks, we add the following set of constraints:

$$\forall e_{i,j} \epsilon E \wedge i \neq j \rightarrow S_i + [\tau \texttt{ of } t_i] + [\texttt{ d of } e_{i,j}] + IT_j \leqslant S_j \quad (7)$$

The constraint states that the start times of task t_i and t_j should allow sufficient time to execute task t_i, to transfer data from task t_i to task t_j and to initiate task t_j.

2) Resource conflict constraint: Two tasks cannot be executed on the same processor at the same time. This is enforced by adding the following set of constraints. MAXINT is a large integer value that exceeds the sum of the execution times of all tasks in the task graph.

$$\forall 0 \leqslant i,j < |T| \ \wedge \ i \neq j \ \wedge \ [\texttt{m of } t_i] = [\texttt{m of } t_j] \ \wedge \ (t_i \texttt{ is not reachable from } t_j) \rightarrow S_i + [\tau \texttt{ of } t_i] + [\texttt{d of } e_{i,j}] + I_{ij} \cdot IT_{ij} \leqslant S_j + (1 - I_{ij}) \cdot \texttt{MAXINT} \quad (8)$$

To reduce the number of ILP variables and equations, we add the condition that t_i *is not reachable from* t_j. It is not necessary to consider Eqn. 8 in this situation because if t_i is reachable from t_j it means that t_i is dependent on t_j and this situation is already covered by Eqn. 7.

3) Other constraints: The next equation enforces positive start times for all tasks.

$$\forall 0 \leqslant i < |T| \rightarrow S_i \geqslant 0 \quad (9)$$

The next equation enforces that one task is only allowed to be scheduled immediately after each task. No task can be scheduled directly after a leaf task in a task graph.

$$\forall 0 \leqslant i,k < |T| \rightarrow \begin{cases} \sum_{j=0}^{|T|-1} I_{ij} \leqslant 1 & \nexists e_{ik} \mid \text{[m of } t_i\text{]} = \text{[m of } t_k\text{]} \\ \sum_{j=0}^{|T|-1} I_{ij} = 1 & \texttt{otherwise} \end{cases} \quad (10)$$

The next equation enforces that one task is only allowed to be scheduled immediately before each task. No task can be scheduled directly before a root task in a task graph.

$$\forall 0 \leqslant i,k < |T| \rightarrow \begin{cases} \sum_{j=0}^{|T|-1} I_{ji} \leqslant 1 & \nexists e_{ki} \mid \text{[m of } t_i\text{]} = \text{[m of } t_k\text{]} \\ \sum_{j=0}^{|T|-1} I_{ji} = 1 & \texttt{otherwise} \end{cases} \quad (11)$$

C. ILP Objective Function

The goal of our optimization is to minimize the completion time of the task graph. Therefore, if we minimize the start time of the last task in the task graph we achieve the goal. A task graph may contain more than one leaf node. In that case, we add a bulk leaf node which has a dependency on all leaf nodes in the original task graph. The objective function is then given by:

$$\texttt{Objective: Minimize } S_{last\ task} \quad (12)$$

VII. Hybrid Prefetch-aware Task Scheduling

In this section, we refine the HybILP scheduler of Sec. VI to consider prefetching. We call the new technique *Hybrid-Prefetch-ILP* (*HybPrefILP*). To attain a compact ILP formulation, we assume that the memory object of only one task can be prefetched during the execution of the running task. This assumption reduces the number of prefetching options which leads to a smaller ILP formulation. As a result of this limiting assumption, the outcome of our scheduler may become sub-optimal, but it is practical. However, two limitations for prefetching often prevent the loading of many memory objects into the local memory. The first limitation is related to the available free space in the local memory. The second limitation comes from the limited time that is available to perform prefetching. Usually, these limitations are barriers to prefetch more than one task in a realistic application. Hence, this assumption does not typically affect the quality of the solution significantly.

We refine the HybILP technique by changing the elements of Eqn. 3. The size of the output data that should be written back to the remote memory (WD_{ij}) is independent from prefetching, but the size of the code (RC_{ij}) and data (RD_{ij}) that must be fetched dependent on the amount of code and data that is already prefetched. We introduce two new constants RC'_{ij} and RD'_{ij} that capture respectively the size of code and data that need to be fetched after the prefetching has ended. These constants replace RC_{ij} and RD_{ij} in Eqn. 3. All other parts of the ILP formulation in the HybILP technique can be used without any change.

A. Size of the code

Assume that task t_j is scheduled after task t_i. The size of the code that needs to be fetched from the memory after the execution of task t_i has ended (i.e. RC'_{ij}) depends, amongst others, on the amount of free space that is left in the local code memory. This code space limitation (CSL) is given by:

$$CSL_{ij} = [\texttt{mc of } pt_{[\texttt{m of } t_j]}] - [\texttt{c of } t_i] \quad (13)$$

Figure 5. Post-processing of a schedule.

RC'_{ij} depends also on the execution time of the previous task as this influences the time during which the DMA and processor can run in parallel (i.e. code can be prefetched). This temporal limitation (TL) is given by:

$$TL_{ij} = \left\lfloor \frac{[\tau \text{ of } t_i]}{H} \right\rfloor \tag{14}$$

We assume that data prefetching is done after code prefetching. The total amount of the code that can be prefetched does not only depend on the space and temporal limitations discussed above. It is also limited by the code size of the task (i.e. RC_{ij} as defined in Eqn. 5). The code prefetched amount (CPA) is given by:

$$CPA_{ij} = \min(RC_{ij}, CSL_{ij}, TL_{ij}) \tag{15}$$

So, the total size of the code that needs to be fetched, when executing task t_j directly after task t_i, is equal to:

$$RC'_{ij} = RC_{ij} - CPA_{ij} \tag{16}$$

B. Size of the input data

Assume once more that task t_j is scheduled after task t_i. The amount of data that needs to be fetched from remote memory to execute task t_j (i.e. RD'_{ij}) depends on the amount of data that could be prefetched during the execution of task t_i. The data prefetched amount depends on the amount of free space in the local data memory. This so-called data space limitation (DSL) is given by:

$$DSL_{ij} = [\text{md of } pt_{[\text{m of } t_j]}] - \sum_{e_{k,i} \in E} [\text{s of } e_{k,i}] - \sum_{e_{i,k} \in E} [\text{s of } e_{i,k}] \tag{17}$$

The data prefetched amount depends also on the running time of the task which is executed directly before task t_j and the amount of the time that the DMA was busy with transferring the code; the latter needs to be considered as we assume that data prefetching is done after code prefetching. This so-called data temporal limitation (DTL) is given by:

$$DTL_{ij} = TL_{ij} - CPA_{ij} \tag{18}$$

The data prefetched amount is limited by the data space limitation, the data temporal limitation, and the actual data memory requirement of the task (i.e. RD_{ij} as defined in Eqn. 6). Hence, the data prefetched amount is equal to:

$$DPA_{ij} = \min(RD_{ij}, DSL_{ij}, DTL_{ij}) \tag{19}$$

So, the total size of the data that needs to be fetched, when executing task t_j directly after task t_i, is equal to:

$$RD'_{ij} = RD_{ij} - DPA_{ij} \tag{20}$$

C. Post-processing

The HybPrefILP scheduler uses the assumption that prefetching of memory objects for task t_j can only be started when its direct predecessor in the schedule (e.g. task t_i) has started. In practice it may sometimes be possible to start the prefetching for task t_j earlier. This may lead to a shorter completion time of the task graph. Consider as an example the schedules shown in Fig. 5. All tasks in this figure are mapped to a single processing tile with 30KB for each of the code and data memories. Schedule A is generated using our HybPrefILP scheduler. This schedule is sub-optimal as the prefetching for task t_2 is only started when task t_1 starts its execution. In practice, it might be possible to start the prefetching of t_2 earlier. Schedule B uses the property that prefetching can be started earlier (i.e. when t_0 is executing). Extending our ILP formulation such that it also considers options in which the prefetching is started earlier would result in a large increase in the number of variables. Therefore, we have decided to keep our current assumption with respect to the start time of prefetching operations. However, we have included a post-processing step that optimizes the schedule. This post-processing step tries to fill the free space of the local memory by prefetching memory object of subsequent tasks based on the task order generated by the HybPrefILP scheduler.

VIII. Experimental Evaluation

The proposed scheduling technique takes an application task graph and MPSoC platform as input. It generates an ILP formulation which is solved using CPLEX [15]. CPLEX is executed on a Linux platform with an Intel® Core^{TM} i7 running at 2.67GHz and 4GB of internal memory. Large execution times are often mentioned as an important drawback of using an ILP-based solution. Thanks to our compact ILP formulation, which avoids unnecessary variables and constraints, the execution time of the ILP solver when looking for a schedule never needed more than a fraction of a minute for the selected task graphs in our experiments. For example, the HybILP technique needs a run-time of 0.76 seconds to schedule an MP3 decoder. The HybPrefILP requires 0.96 seconds to schedule the same application while also considering prefetching.

We use two different task graphs to evaluate the proposed scheduling technique. The first task graph models an MP3 decoder. This application is a frequently used application from the multimedia domain. We manually extract a task graph for this application (see Fig. 6). We estimate the execution time of all tasks when they would be executed on an ARM7TDMI core. The second task graph is taken from [12]. It is a synthetic directed acyclic graph with 50 tasks. In our experiments, we assume that the latency of the remote memory is 100 times larger than the local memories and access to the local memory will take one clock cycle.

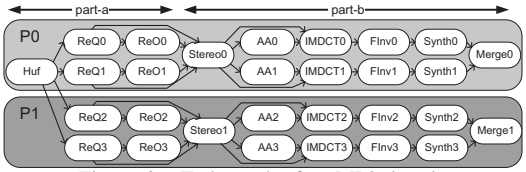


Figure 6. Task graph of an MP3 decoder.

A. MP3 Decoder

MP3 decoding is a frame based algorithm that transforms a compressed stream of data into pulse code modulation (PCM) data. Fig. 6 shows the task graph of the MP3 decoder. As the task graph of this application is composed of two symmetric sub-graphs, we map each sub-graph onto a single processor. This mapping is depicted in Fig. 6; the upper sub-graph is mapped to the first processor (P0) and the lower sub-graph is mapped to the second processor (P1).

To make a comparison between our technique and related work, we implement the greedy CPU (G-CPU) scheduling technique from [12]. We also implement the *heterogeneous earliest finish time* (HEFT) scheduling technique [16] which is a commonly used heuristic-based mapping and scheduling technique. We apply the HybILP, HybPrefILP, HEFT and G-CPU scheduling techniques to the task graph of the MP3 decoder. The HEFT and G-CPU result in the same outcome; the result of the HybILP and HybPrefILP scheduling is different from these techniques. HEFT and G-CPU schedule all tasks of the MP3 decoder in a code-driven order. The HybILP and HybPrefILP schedule part-a of the task graph (shown with an arrow in Fig. 6) in a data-driven order. In part-a the size of the intermediate data is larger than the code size of the tasks. By using a data-driven strategy, the data outputted by one task is consumed immediately by the next task. Therefore, there is no need to store/load this large intermediate data to/from the remote memory. The code size of the tasks are larger than the size of the intermediate data in part-b of the MP3 decoder task graph. The HybILP and HybPrefILP schedule part-b in a code-driven order. This decision leads to a reduction in the number of off-chip memory accesses because with the code-driven strategy each task only needs to be loaded once from the remote memory. As a result of these scheduling strategy decisions, HybILP achieves a schedule with a 7% shorter execution time (in terms of cpu cycles) as compared to HEFT and G-CPU. The HybPrefILP technique finds a schedule with an 8% shorter execution time as compared to extended versions of HEFT and G-CPU in which prefetching is considered in a post-processing step. These results show that our scheduling technique is able to construct schedules that are significantly faster as compared to existing well-known scheduling techniques. The schedule constructed by our hybrid prefetch-aware technique is 11% faster as compared to a the non-prefetch-aware schedule constructed by our technique. This shows the advantage of using prefetching to decrease the run-time of applications.

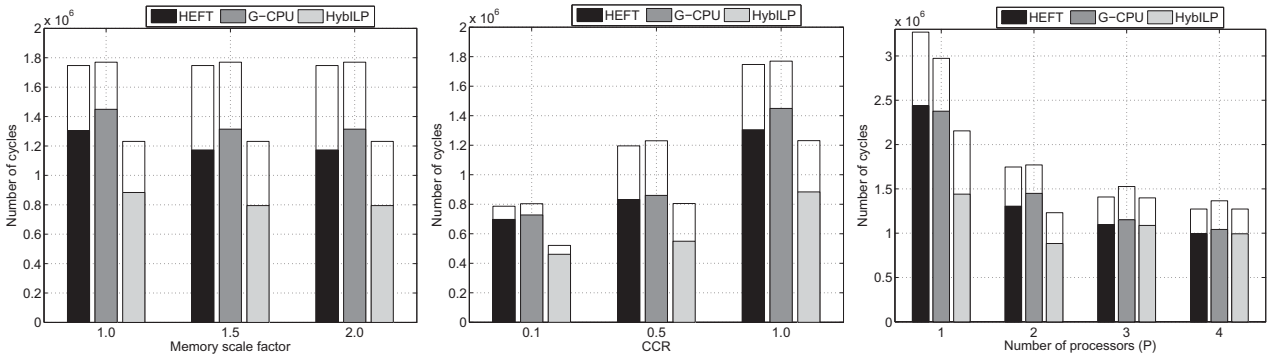
B. Synthetic DAG

To verify the effectiveness of our proposed technique we select the synthetic task graph used in [12]. The mapping decisions computed by HEFT are used as input to our scheduling technique. We evaluate our technique for three different experimental set-ups: first, different sizes of local memories; second, different amounts of communication-to-computation ratios (CCRs) in the task graph; third, different numbers of processors in the platform.

The size of the local memory can affect the amount of prefetching. We determine the necessary amount of local memory which is required to execute the task graph. Fig. 7(a) shows execution time of the task graph (in the number of processor cycles) for different sizes of the local memory, for two processors, and a CCR of 1.0. The memory scale factor in Fig. 7(a) is the scaling factor of the necessary amount of the local memory. Scaling local memory size from 1.0 to 1.5 times decreases the run-time of the application when using HybPrefILP by 10%, when using HEFT with prefetching by10%, and when using G-CPU with prefetching by 9%. Scaling local memory size from 1.5 to 2.0 times does not further reduce the run-time of the application. This is due to another limitation of prefetching which is the time limitation (see Eqn. 14). These results show that the effect of prefetching is similar in all these three scheduling techniques. However, overall, the HybPrefILP gives a schedule which is 32% and 39% faster compared to HEFT with prefetching and G-CPU with prefetching respectively for all local memory scaling factors in Fig. 7(a).

The number of off-chip memory access determines the required amount of communication in a multiprocessor system. Hence, we explore the effectiveness of our proposed scheduling (HybILP) for different amounts of communication to computation ratios. For this purpose, we derive several task graphs from the original task graph by scaling the size of the memory objects. CCR is a term for digitizing the amount of communication in an application. A larger CCR implies a larger amount of remote memory accesses. Fig. 7(b) shows the execution time of the task graph (in the number of processor cycles) for different CCRs when the task graph is mapped to a platform with two processors and the memory scale factor is 1.0. The required amount of processor cycles increases by increasing the CCR of the task graph. This is due to the fact that more remote accesses are needed in a task graph with larger CCR. Comparing the outcome of HybILP (HybPrefILP) with the outcome of HEFT (HEFT with prefetching) and G-CPU (G-CPU with prefetching) in Fig. 7(b) confirms that our technique outperforms common heuristic techniques in different CCRs.

By increasing the number of the processing tiles in the MPSoC, the required amount of the processor cycles decreases (see Fig. 7(c)). This shows the existence of parallel tasks in the task graph that enable tasks to execute in a concurrent way on a multiprocessor system. For the selected DAG, the HybILP (HybPrefILP) gives similar efficiency to HEFT (HEFT with prefetching) and G-CPU (G-CPU with prefetching) by using a platform with fewer processors. For example, the execution time of the HybILP schedule on a platform with two processors is close to the execution time of HEFT and G-CPU schedules on a platform with four processors. This means that the HybILP requires less computation resources compared to the HEFT and G-CPU.



(a) configuration [2-1.0-X] (b) configuration [2-X-1.0] (c) configuration [X-1.0-1.0]

Figure 7. Execution time of a DAG in different configurations ([# of processors][CCR][memory scale factor]). Each filled bar shows the execution time of a technique when considering prefetching and the stacked bar represents the extra execution time of the same technique without prefetching.

Increasing the number of processors up to three (or four) does not reduce the execution time of the task graph significantly while using the HybILP (HybPrefILP) for scheduling the selected DAG; this is due to the limited parallelism in the DAG and on-chip communication overhead.

The outcome of the proposed technique for the selected DAG is a hybrid schedule (i.e. it uses the combination of code- and data-driven scheduling for tasks in the task graph) which reduces the amount of remote accesses for large code elements and large intermediate data elements between the tasks. When comparing the outcome of HybILP with HEFT and G-CPU, our technique achieves a 29% and 30% respectively shorter execution time for the selected DAG in a nominal experimental configuration (where two processors, a CCR equal to 1.0, a memory scale factor equal to 1.0). By extending HybILP to HybPrefILP, the execution time of the DAG in the nominal experimental configuration reduces by 32% and 39% compared to HEFT with prefetching and G-CPU with prefetching respectively.

IX. Conclusion

The performance of applications when running on an MPSoC are affected by the time spent on fetching code and data from remote memories. We present a scheduling technique to improve the memory behavior of an MPSoC with limited on-chip memories. We formulate our approaches using an ILP framework.

The proposed technique, HybILP, makes a trade-off between loading code or data to reduce the run-time of the application. In essence, it chooses the most suitable scheduling strategy for a series of tasks in a task graph. Tasks with dominant code size are scheduled with a code-driven scheduling strategy and tasks that exchange large amounts of data are scheduled with a data-driven scheduling strategy. Our technique uses a compact ILP formulation which requires limited time for an ILP solver. To further refine the result of HybILP, we extend it to HybPrefILP. The HybPrefILP technique takes the overhead of prefetching into account when scheduling an application onto an MPSoC.

We evaluate our scheduling technique with a synthetic task graph, taken from recent related work [12], and a common multimedia application (an MP3 decoder). We achieve a reduction in run-time of 8% compared to a common heuristic solution for the MP3 decoder application and 32% for the task graph from [12]. These results demonstrate the advantage of our hybrid prefetch-aware scheduling technique.

References

[1] D. A. Patterson *et al.*, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[2] R. Banakar *et al.*, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," *CODES*. ACM, 2002, pp. 73–78.

[3] M. Gallet *et al.*, "Efficient scheduling of task graph collections on heterogeneous resources," *IPDPS*. IEEE, 2009, pp. 1–11.

[4] J. Hu *et al.*, "Energy- and performance-aware mapping for regular noc architectures," *IEEE Trans. on Comp.-Aided Des. Integ. Cir. Sys.*, vol. 24, no. 4, pp. 551–562, 2005.

[5] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, pp. 355–383, 2003.

[6] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," *DAC*. ACM, 2007, pp. 777–782.

[7] H. Yang *et al.*, "Pipelined data parallel task mapping/scheduling technique for MPSoC," *DATE*. IEEE, 2009, pp. 69–74.

[8] S. Kwon *et al.*, "Data-parallel code generation from synchronous dataflow specification of multimedia applications," *ESTIMedia*. IEEE, 2007, pp. 91–96.

[9] F. Chen *et al.*, "Optimizing overall loop schedules using prefetching and partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, pp. 604–614, 2000.

[10] L. Zhang *et al.*, "Variable partitioning and scheduling for MPSoC with virtually shared scratch pad memory," *J. Signal Process. Syst.*, vol. 58, pp. 247–265, 2010.

[11] W. Hongmei *et al.*, "Dynamic management of scratchpad memory based on compiler driven approach," *ICCSIT*, 2010, pp. 668–672.

[12] M. Gallet *et al.*, "Scheduling complex streaming applications on the cell processor," *IPDPS*, 2010, pp. 1–8.

[13] H. Chang *et al.*, "Access-pattern-aware on-chip memory allocation for simd processors," *IEEE Trans. on Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, pp. 158–163, 2009.

[14] L. De-feng *et al.*, "Multi-bank memory access scheduler and scalability," *ICCET*, 2010, pp. 723–727.

[15] ILOG CPLEX: High-performance software for mathematical programming and optimization, 1997-2008. url: http://www.ilog.com/products/cplex/

[16] H. Topcuoglu *et al.*, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, 2002.